DICTA2002: Digital Image Computing Techniques and Applications, 21—22 January 2002, Melbourne, Australia.

Programming Optimisation for Embedded Vision

Aster Wardhani and Binh Pham
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434 Brisbane Q4001 Australia
{a.wardhani; b.pham@qut.edu.au}

Abstract

Increasing demand in mobile computing and multimedia market as well as other embedded devices has made it necessary to develop computer vision techniques which are more appropriate for these systems. This paper firstly analyses requirements for embedded vision systems and issues involved in programming optimisation to meet these requirements. Two case studies are then presented together with the analysis of results.

1. Introduction

Embedded systems are steadily growing due to increasing demand in areas such as mobile computing, autonomous robots, and multimedia. These systems have a number of common constraints: low cost, low power, limited amount of on-board memory and real time operation. Thus, some efforts have been devoted to designing special purpose chips [1] and to optimise compilers [2] to meet these requirements.

Traditional image processing and computer vision algorithms often require a large amount of storage and massive computing power to obtain real-time performance. For applications such as autonomous robots where constant streams of information from the real world need to be perceived by sensors in order to determine or plan a course of action and to update the world model, a few approaches have been used: attaching a computer to a robot; designing a custom chip for specific vision tasks; processing images off-line; or re-designing these algorithms and optimising the programs to increase performance and reduce the size of code and memory required.

This paper focuses on the last approach. We first examine important issues to be considered in re-designing and optimising algorithms for embedded vision systems. The discussion covers the adaptation of compilers optimisation techniques, performance versus code size, and strategies for power reduction. We then present a number of optimising strategies and analyse the results for two case studies of common computer vision tasks.

2. Issues in optimising algorithms and programs for embedded systems

Computer vision algorithms cover a broad spectrum from low-level processing (e.g. edge detection to control reactive behaviour in robots) to high-level tasks such as symbolic processing and planning (e.g. navigation, path finding, obstacle avoidance, learning and group behaviours). They require vast amount of memory to store input and interim images that are required for analysis. Yet, the storage of whole images is not necessary in many cases. For example, it is only necessary to compute optical flow for a strip of an image in order to detect an obstacle [3].

Traditional computer vision algorithms are also computationally expensive because large amounts of processing are required to operate on large amounts of data. To ensure their robustness, most algorithms generally cover all special cases, even though these special cases might not occur often. Thus, there is scope for removing some redundancy. To cater for embedded vision, we should therefore aim to adopt a minimalist approach where only an absolutely minimum amount of work would be performed on a minimum amount of data in order to achieve a specific task.

Another characteristic worth noting is a common practice to sequentially process computer vision tasks in a pipeline fashion. For example, convolution is performed then optical flow, or segmentation is performed then contour following and object recognition. However, there are a few drawbacks in this simple approach. Decisions made for each task are carried out independently of those for another task; hence it is not possible to take advantage of relevant knowledge across tasks. Furthermore, no decision can be made for a specific task until previous tasks are completed. This may cause delay in obtaining results (e.g. to allow a course of action for an autonomous robot to be determined in real time). These drawbacks may be overcome by investigating a more integrated approach where only a portion of the first task is performed to obtain sufficient information for a portion of the second task being carried out, before another portion of the first task is performed, and so on. However, we only deal with problems involving single task in this paper while leaving multi-tasks problems as the subject of another paper.

Current compiler technology does not meet the demand for producing an optimised computer vision code for embedded systems. In particular, this holds for the DSP processors. The largest part of DSP software is still developed manually at the assembly level [4]. However, the earlier approach of using assembly languages to improve performance has been found not economical because of the high cost of development and maintenance and the lack of re-usability and portability. This approach is also not suitable for embedded systems because of the large code density. Later attempts therefore used high-level languages such as C or C++ as the main languages for embedded processors [4]. We also adopt this approach.

A study by Nachtergaele et al.[5] has found that the power consumption in signal processing systems is dominated by memory access and global communication. Thus, in re-designing algorithms for embedded vision systems, we need to examine ways to minimise memory locations and data transfer (e.g. organising data so that computation can be carried out locally).

3. Optimisation Strategies

There are many common programming optimisation strategies ranging from improving the order of complexity, control flow, to low-level strategies such as loop unrolling and loop hoisting (to take redundancies from inner loops to outer loops). Optimisation techniques for compilers that have been used for most cases have a drawback because they aim mainly at improving performance without considering code size [6,7]. Many existing C compiler with DSP chips, however, produce large code size [8]. We only discuss here those strategies that are most relevant and exert significant improvement on computer vision programs. To perform optimisation the following steps are required:

- Selection of an efficient algorithm: in computer vision, there are many existing techniques to achieve certain tasks. A careful selection is required to choose the most efficient algorithm in order to achieve a better performance.
- Identifying the bottlenecks: optimisations are focused on areas in the code that contribute to the largest processing time.
- Code re-writing: to minimise the amount of unnecessary optimisations, the choice of appropriate data structures and a simple and clear code is important.

4. Case Studies

In this section the high-level optimisation strategies are applied to two commonly used image processing algorithms: (1) edge detection; and (2) optic flow detection[1].

4.1 Edge Detection

In general this algorithm can be described into three main loops:

1. Read input image
2. Convolution with the horizontal and vertical masks
3. Write output image

To analyse the bottleneck, the following performance table is obtained.

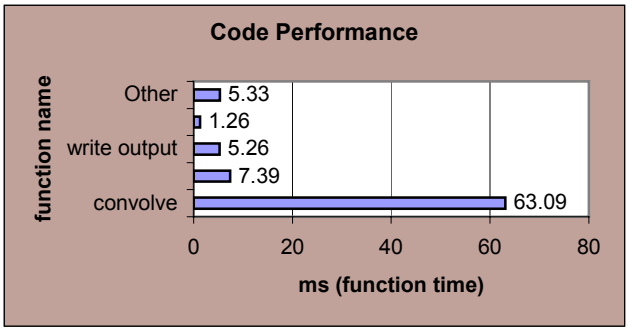


Figure 1 Performance of edge detection algorithm

Figure 1 shows that the convolution function produces the most computations taking 63.09 ms. The C code for the convolution is illustrated as follows.

```
void convolve_filter(int result[H][W], int
image[H][W], int mask[2*N+1][2*N+1])
{
   const int N = 1;

   for j = 0 to H-1 do {
    for i = 0 to W-1 do {
      sum = 0;
      for y = -N to N do {
        for x = -N to N do {
          sum += image[N+j+y][N+i+x] *
                 mask[N+y][N+x];
        }
      }
      result[j][i] = sum;
    }
   }
}

this is called twice:

convolve_filter(grad_x, input_image,
  sobel_mask_x);
convolve_filter(grad_y, input_image,
```

Figure 2. Edge Detection Code

[1] Experiment results are obtained with Pentium III and 128 Mb RAM PC.

In this experiment, Sobel detector is used (N=1). The image dimension is HxW. Two extra images, *grad_x* and *grad_y*, are required to store the gradient magnitudes results. Sobel masks are defined by using two arrays of constants. To reduce this bottleneck the following optimisation is performed.

- *Dead code elimination*

The first instance in optimising the code is to eliminate all the unnecessary variables and operations. The use of this technique is explained together with the loop folding technique below.

- *Loop transformation*

Loops contribute most bottlenecks in image processing applications, thus loop transformation is often the most effective means of code optimisation. The most common usage is the *loop unrolling* and *loop folding*. In loop unrolling, the statement inside the loop is duplicated, reducing the number of branching tests required. Unrolling the loop completely will eliminate the branching and testing entirely. Loop unrolling also provides a higher potential of parallelisation of instructions during scheduling, at the cost of an increase in code size. In the edge detection example, unrolling the loop can be performed in the inner loop as follows.

```
for y = -N to N do {
   sum += image[N+j+y][i-1] * mask[N+y][0];
   sum += image[N+j+y][i]   * mask[N+y][1];
   sum += image[N+j+y][i+1] * mask[N+y][2];
}
```

Figure 3. Optimisation 1: Loop unrolling (iter = 3).

Since in this case N = 1, the maximum number which the inner statement can be iterated is 3 times. Unrolling the loop completely, resulting in the following code.

```
sum = image[j-1][i-1]*mask[0][0] + image[j-1][i]
    * mask[1][0]+image[j-1][i+1]*mask[2][0] +
    image[j][i-1]*mask[0][1] + image[j][i]
    * mask[1][1] + image[j][i+1]* mask[2][1] +
    image[j+1][i-1]*mask[0][2] + image[j+1][i]
    * mask[1][2] + image[j+1][i+1] * mask[2][2];
```

Figure 4. Optimisation 2: Loop unrolling (iter = complete).

The loop folding combines adjacent loops, which loop over the same range of the same variable. The incrementing and testing of the iterator is performed only half as often. Under some circumstances, some variables can be eliminated, improving the locality of reference and cache behaviour. In the edge detection case, the convolution is called twice to calculate the two directional gradients. These two loops can be merged reducing the need to store each result temporarily (dead code elimination). This optimisation is illustrated in Figure 5. The dead code elimination is applied by taking out the extra arrays required to store the intermediate values of the edge gradients (previously called *grad_x*, *grad_y*).

```
const int N = 1;
int sobel_y[3][3] = {-1, 0, 1, -2, 0,
                      2, -1, 0, 1};
int sobel_x[3][3] = {1, 2, 1, 0, 0,
                      0, -1, -2, -1};

for j = 0 to H-1 do {
  for i = 0 to W-1 do {
    y_sum = x_sum = 0;
    for y = -N to N do {
      for x = -N to N do {
        y_sum += image[N+j+y][N+i+x] *
                 sobel_y[N+y][N+x];
        x_sum += image[N+j+y][N+i+x] *
                 sobel_x[N+y][N+x];
        }
      }
      mag = max(y_sum, x_sum);
      if (mag > threshold)
       out_image[j][i] = 0;
      else
       out_image[j][i] = 255;
   }
```

Figure 5. Optimisation 3: Loop folding and dead code elimination

- *The use of constants, look-up table and strength reduction*

Often, lookup tables can produce better performance, in particular for iterative and recursive computations or when the range of values is known and not large. The Sobel masks can be replaced by constants as follows.

```
y_sum = -image[j-1][i-1]-(image[j-1][i] << 1)
  -image[j-1][i+1] + image[j+1][i-1]
  +(image[j+1][i] << 1) + image[j+1][i+1];

x_sum = image[j-1][i-1] - image[j-1][i+1]
  +(image[j][i-1] << 1) - (image[j][i+1] << 1)
  + image[j+1][i-1] - image[j+1][i+1];
```

Figure 6. Optimisation 4: using constant

Strength reduction is the replacement of an expression by others that yield the same value but is less expensive to compute. Some compilers performed this optimisation. In the edge detection algorithm, the multiplication of the mask coefficient with the image pixel can be replaced with the shift operation as follows.

```
y_sum = - image[j-1][i-1] - 2*image[j-1][i]
        - image[j-1][i+1] + image[j+1][i-1]
        + 2*image[j+1][i] + image[j+1][i+1];

x_sum =   image[j-1][i-1] - image[j-1][i+1]
      + 2 * image[j][i-1] - 2*image[j][i+1]
      + image[j+1][i-1] - image[j+1][i+1];
```

Figure 7. Strength reduction

- *Memory reduction*

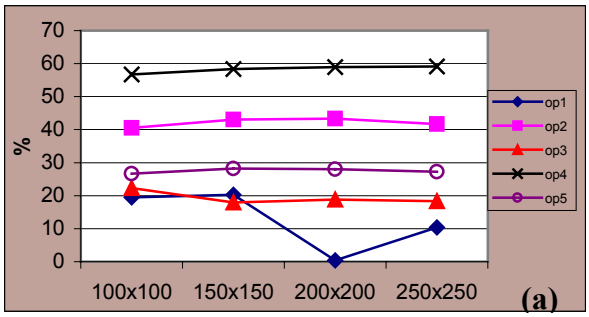
For simplicity, often an image whose size is H x W is implemented as a 2D array of that size. Most vision

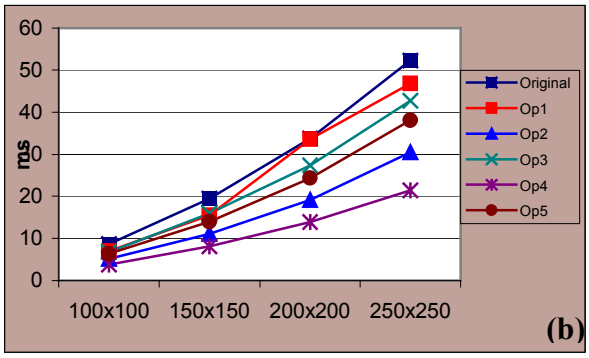
algorithms contain a convolution-based code, which operates on a neighborhood (N x N) of pixel values at an instance of time. At this instance, not all pixels of size H x W are required, thus, there is an opportunity for reducing the amount of memory used to just providing pixel values required at one operation at a time and updates the array for the next operation at the appropriate time. A circular buffer idea was used in [9]. Reducing the memory usage can also improve the power consumption of the embedded systems [5,9]. In a wavefront processing [10], data is collected from the array line-by-line in the flight direction. The data is processed from the upper left corner of the image, working downward and rightward. This wavefront and other block based processing are possible techniques in reducing the memory usage.

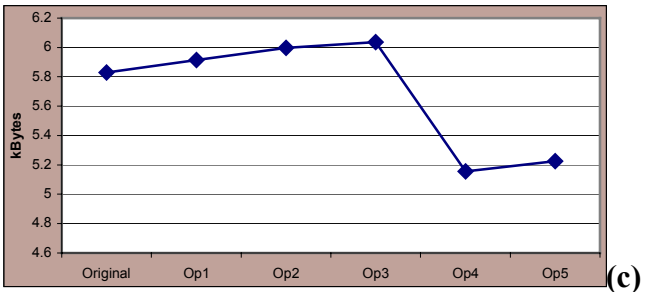
In this paper, the circular buffer idea is also used. Initially three lines (N=1) are read to cover the convolution at [j-1][i-1] and [j+1][i+1]. After the entire row has been convolved, a new row of image is read into an array.

The optimisations described above were applied and yield some improvement to the execution time of the edge detection code. Not all of the optimisation strategies yield to a drastic change of code performance. For example, the use of constant and strength reduction, only improve the speed by a small amount. To illustrate the improvement contributed by the above optimisations, the following results were obtained:

- *Op1: loop unrolling (iteration = 3)*
- *Op2: complete loop unrolling*
- *Op3: loop folding and dead code elimination*
- *Op4: complete loop unrolling, constant and strength reduction*
- *Op5: in-place optimisation*







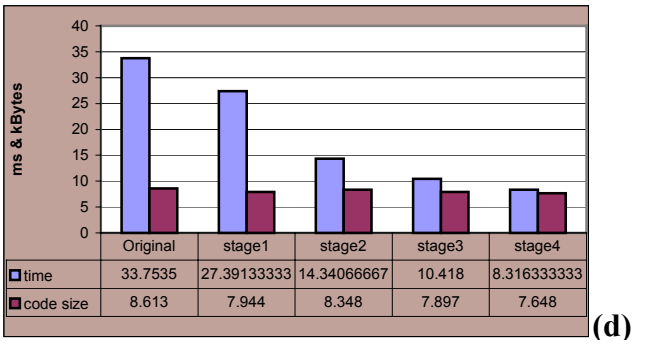


Figure 8. Optimisation Results: (a) % of improvement; (b) execution time; (c) code size; and (d) overall

Results show that all optimisations improve the execution time of the edge detection code (Figure 8(b)). In Figure 8(a), *Op4* contributes to the highest improvement. *Op1* has a variant amount of improvement at different image sizes. This and other optimisations that involve loop unrolling, also contribute to a small increase in code size (Figure 8(c)). To test the overall improvement, *Op3*, *Op4,* and *Op5* are all applied to the code (*Op4* represents both *Op2* and *Op1*, since it contains a complete loop unrolling (*Op2)* and the complete loop unrolling is chosen since it has better performance than incomplete loop unrolling (*Op1)*). The overall improvement is shown in Figure 18(d). This diagram shows that the overall optimisation can achieve a reduction in execution time from 33.75 ms to 8.32 ms with a small increment of code size.

4.2. Optical Flow

Optical flow calculation has been known to be computationally expensive. In general the gradient-based approaches [11] are faster than the region matching [12] based approaches [13]. However, with the recent work in the minimalist optic flow techniques [3], the region matching approach can be used with a relatively fast and accurate implementation. To demonstrate the optimisation process of an optic flow algorithm, this approach is selected as the appropriate choice. One minimalist method proposed by Young et. al. [3] performs an optic flow calculation in a subset of area in the image, where objects are likely to be present. This area is then divided into

blocks. The optic flow estimate for each block is calculated using the region matching approach. Ancona and Poggio [14] proposed some methods to reduce the matching searching space. Rather than searching the entire N x N directional possibilities, where N is the searching window size, the minimum value at x and y directions can be found by finding the minimum at one direction, say x, which becomes the minimum in x, then at this location, find the minimum in y direction. Thus, the searching area is reduced from a rectangular area into a "cross" area. Applying the region matching at each block yields the approximate optic flow field at the pre-selected area in the image. The coarse to fine hierarchical searching method proposed by Giachetti [15] further increase the speed and accuracy of the searching method.

The minimalist approach to optic flow calculation can then be summarised as follows.

1. Select a strip in the image as an area of interest.
2. Divide the strip into blocks
3. For each block calculate the optic flow ***of(x,y)***
    a. Find the minimum at the vertical searching direction = $\mathbf{of}_y$
    b. At this minimum location, find the horizontal minimum = $\mathbf{of}_x$

The following describes this step in C program. The code above is presented slightly different to the actual implementation to conserve space.

```
int find_minima(Direction dir, int x0, int
y0, int initLoc)
{
   while(step_sz >= 1)  {
for (loc = start; loc < end; loc += step_sz) {
      if (dir == e_verti)
        val = do_match(x0,y0,initLoc,loc);
      else
        val = do_match(x0,y0,loc,initLoc);
      if (val < minVal) {
        minVal = val;
        minLoc = loc;
      };
    };
    reduce_step_size(&step_sz, &start,&end);
    return(minLoc);
   };
};
double do_match(int x1, int y1, int x2, int
y2){
  for(j=0; j < Y_BLOCK_SZ; j++) {
    for(i=0; i < X_BLOCK_SZ; i++) {
        res += fabs(get_pixel(x1+I,y1+j,time)
                - get_pixel(x2+I,y2+j,time+1);
     };
  };
  return(res);
};
for(i = 0; i < NUM_BLOCKS; i++)
{
  op_x[i] =
    find_minima(e_horiz,c_x,c_y,x_init);
  op_y[i] =
    find_minima(e_verti,op_x,c_y,op_x[i]);
};
```

Figure 9. Minimalist optic flow code

In the function *find_minima, reduce_step_size* is added and computes the new step size and the new coarse to fine searching start and end points. Initially, *step_sz* is set to be eight and it is decremented until *step_sz* equals to one. The function *do_match* calculates the matching correlation between block areas in the current and next frame. The pixel value at certain location in the frame is obtained by the *get_pixel* function. The block size used is set to be *Y_BLOCK_SZ* = 20 pixels height and *X_BLOCK_SZ* = 30 pixels wide. For each optic flow calculation, find_minima is called twice, one for the x direction and one for y. The optic flow calculation is performed at *NUM_BLOCKS* times and this value is set depending on the width of the image. An example of optic flow results are described as follows using the portion of theYosemite image sequence[2] (frame 9 and 11).

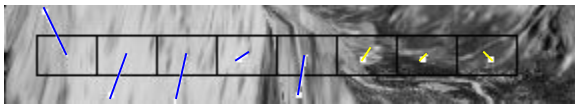


Figure 10. Example of minimalist optic flow results

Profiling the above code, shows that the function *do_match* contributes the highest bottleneck, thus requires some optimisations. This function takes approximately 218.9 ms to run using the image size of 316 x 252. Two optimisation strategies were implemented:

- *Dead code elimination:*

Initially for modularity and clarity, pixel values are accessed through the call of an inline function called get_pixel. This function however can be eliminated and a direct access to a three dimensional array, storing pixel values at each frame, is used instead. This yields the following code for the inner loop statement.

```
res += fabs(image[x1+i][y1+j][time] -
       image[x2+i][y2+j][time+1]);
```

Figure 11. Dead code elimination on the matching function

Applying this modification, the time it takes to execute do_match function reduces to 39.69 ms.

- *Loop unrolling:*

The above inner loop statement can be iterated to reduce the execution time further. The number of iteration chosen for the selected block size is 2, 5, and 10. The optimisation results are shown below.

[2] The test data is obtained from ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/ used for optic flow benchmark, compiled by John Barron [13]

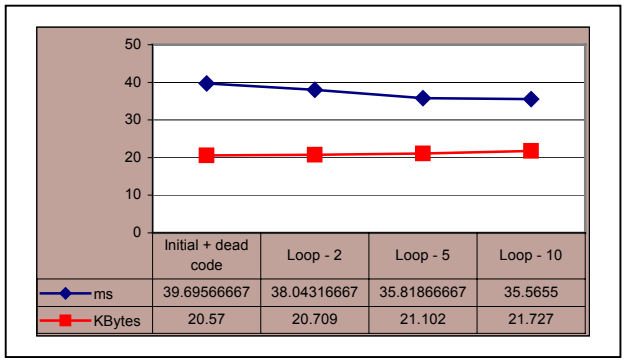


Figure 12 (a). Loop unrolling results

Figure 12(a) shows results after the dead code elimination strategies applied to the code. This code is then unrolled in three different iterations. As the iteration increases, the code becomes slightly faster and the code size becomes larger.

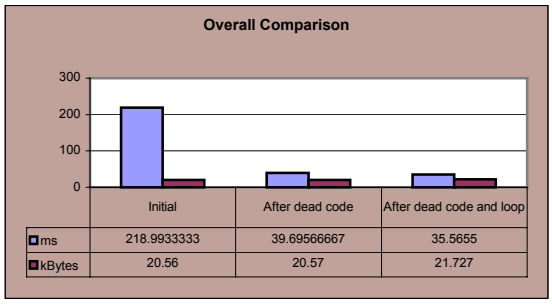


Figure 12(b). Overall comparison.

Figure 12(b) summarises the profiling results between initial code, code after dead code elimination and code with the largest loop unrolling. The diagram shows a significant speed improvement with a very little increment in code size.

5. Conclusion and Future Work

We have described our early experiment on optimising code for embedded targets. The results show that the optimisation techniques applied to the two case examples can reduce the computation time by a large amount. The code size is currently has not impose a significant change.

What we have achieved so far is only limited to local code optimisations and we have not yet consider the more global optimisation which involve re-organisation of the structure of the algorithm and/or combining one algorithm with another. By re-arranging a group of algorithms such as ones that are involved in vision tasks can reduce the amount of code and data redundancy and open to a new and smarter way of producing results cheaper and faster. We leave this as a future work.

6. References

[1] A. Sudarsanam, S. Malik, S. Rajan, and M. Fujita, "Development of a high-quality compiler for a Fujitsu fixed-point digital signal processor," *Proceedings of the Seventh International Workshop on Hardware/Software Codesign ACM SIGDA*, pages 2-7, May 1999.

[2] S.Liao, S.Devadas, K.Keutzer, S.W.K. Tjiang, and A.Wang, "Code optimization techniques for embedded DSP microprocessors," *Design Automation Conference*, pp.599-604, 1995.

[3] G.Young, M. Herman, T. Hong, D. Jiang and J. Yang,, "New Visual Invariant for Terrain Navigation without 3D Reconstruction", *International Journal of Computer Vision*, 28(1), pp. 45-71, 1998.

[4]C. Schläger and V. Zivojnovic, "C/C++ based techniques for production-quality DSP code development," *Proceedings of ICSPAT'95*, 1995.

[5] L. Nachtergaele, V. Tiwari, N. Dutt, "System and architecture-level power reduction of microprocessor-based communication and multi-media applications," *IEEE/ACM International Conference on Computer Aided Design. ICCAD'2000*, IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140), pp.569-73, 2000.

[6] M.Levy, "C Compilers for DSPs flex their muscles," *EDN Access*, Issue 12, 1997.

[7] R. Leupers, "Code Generation for Embedded Processors," *Proceedings 13th International Symposium on System Synthesis (ISSS'00)*, pp. 173-179, 2000.

[8] V. Zivojnovic, H. Schraut, M. Willems and R. Schoenen, "DSPs, GPPs, and multimedia applications: An evaluation using DSPstone," *Proceedings of ICSPAT'95*,1995.

[9] L. Nachtergaele, D. Moolenaar, B. Vanhoof, F. Catthoor, H. De Man, "System-level power optimization of video codecs on embedded cores: a systematic approach," *Journal of VLSI Signal Processing*, Kluwer, Boston, Vol. 18, No. 2, pp. 89-111, February 1998.

[10] C. Coelho, N. Roma and L. Sousa, "Pipeline Architectures for Computing 2-D Image Moments," *XIV Conference on Design of Circuits and Integrated Systems DCIS'99*, 1999.

[11] Lucas, B. D. and Kanade, T., "An Iterative Image Registration Technique with an Application to Stereo Vision", Proceedings of the DARPA Image Understanding Workshop, pp.121-130, 1981.

[12]T. Camus, "Real-Time Quantized Optical Flow", *Proceedings of IEEE Conference on Computer Architectures for 18 Machine Perception*, Como, Italy, 1995.

[13] J.L. Barron, D.J. Fleet and S.S Beauchemin, "Performance of Optical Flow Techniques", *International Journal of Computer Vision*, Vol. 12, no. 1, pp. 43-77, 1994.

[14] N. Ancona and T. Poggio, "Optical Flow from 1-D Correlation: Applications to a Simple Time to Crash Detector," *Proceedings of IEEE International Conference on Computer Vision*, pp. 209-214, 1993.

[15] A. Giachetti, "Matching Techniques to compute image motion," *Image and Vision Computing,* Vol.18. No.3, pp.245-258, 2000.